FIT2099 - Assignment 3

Design Rationale

👉Only for new things we added this assignment and old things we changed (refactored) from previous assignments.
⭐See page no. 6 for a table that maps displayChar to the object it represents
⭐See page no. 7 to see which branches in git was used for which design features

Members:

Abhishek Shrestha
Amindu Kaushal

Objective: Explain how the system will work. Describe reasoning behind design decisions

Design Principles Used:

1. Avoid Excessive use of literals
2. Don't repeat yourself (DRY)
3. Classes should be responsible for their own properties
4. Reduce dependencies as much as possible (ReD)
5. Declare things in the tightest possible scope
6. Delegation over inheritance
7. Command query principle
8. Liskov design principle

Second Map

To implement all the requirements for a second map, we realized that we had to change the code in the *GameMap* class. However, since the *GameMap* class was part of the engine, we were not allowed to edit it. Thus, we created a *new ParkGameMap* class that **extends** *GameMap* from the engine, allowing us to reuse code and not repeat ourselves (DRY). We added all the possible *groundTypes* to the second map as well. In order to place the second map exactly north of the first, we created a new *addExitFromHereToAnotherMap()* function in the *ParkGameMap* class, that does exactly what the name suggests - adds an exit to another map, which allows the player to move(exit) from one map to another. Since this was all we needed, we were able to declare things in the tightest possible scope. Though both maps are now declared as instances of the *ParkGameMap*, we could add it to the *Arraylist* of type *GameMap* (in the *World* class) without any issues. That is because we are passing in a child class of *GameMap* (*ParkGameMap* is a *GameMap*), hence following the Liskov Design Principle.

Lakes, Water and Rain

We **extended** the *Ground* class to make the specified *Lake* class (Example of DRY). We overrode the tick() method so that the number of fish in a Lake could increment by 1 for a 60% chance each turn. We set a max limit of sips in a Lake to be 25. If a lake does not have any sips, then it is dry, and hence should not have any fish (if *sips* == 0, *noOfFish* = 0). Similarly, we added public getters and setters for only attributes that the rest of the game package needed access too. We have scarcely used getters and setters in most of our classes, following the Command query principle.

In order to allow Pterodactyls to fly over Lakes, we created a Flight **Enum** class, and added it as a **capability** to the Pterodactyl in its constructor. Lakes only allow actors who have this Flight capability (Pterodactyl).

We implemented rainfall in the ParkGameMap class by **overriding** the tick() function. We made it so that every 10 turns, with a probability of 20% the sky would rain, adding some amount of sips to each Lake. This is achieved by calling the getter and setter for sips from the Lake class. Since sips is park of the Lake class, this follows the classes should be responsible for their own properties principle.

We figured instead of using Fish objects, we can use an int to store the number of fish in each Lake as an instance variable. This allowed us to reduce dependencies (ReD).

Searching the map

Since we often found ourselves searching for either an item(fruits, corpses), ground(lake, bushes, trees) or an actor(pterodactyl, stegosaur) we have added a seperate class called SearchMap that iterates through the entire map and looks for the target object. This process of looking through the map has been used in the implementation of multiple design features such as

- SeekFoodBehaviour : Different types of dino's looking for food they eat
- SeekWaterBehaviour : Dinos searching for water when thirsty
- PterodactylSeekTreeBehaviour : Pterodactyls looking for trees for resting/laying eggs

Therefore using this class allowed us to remove an extensive amount of repeating code(DrY). Similarly, since all map searching would only be handled by this class, it allowed us to follow: Classes should be responsible for their own properties.

Thirsty Dinosaurs

To implement thirst, we have added an instance variable to store the current water level of the dinosaur which reduces by one each turn. And to keep track of unconsciousness we have added a separate counter for unconsciousness due to lack of water. We have **reused** the isConscious() method from Actor class by **overriding** it and updating it to return true if only both food and water levels are above 0. Regarding dinosaurs searching for water when thirsty, we have added a SeekWaterBehaviour class that implements the **Behaviour** interface (use of interfaces is an example of delegation over inheritance) and utilizes the

SearchMap class (DrY) to look for a lake and returns a MoveActorToConsumeAction which moves the dinosaur towards the closest lake.

Pterodactyls

Since Pterodactyls should be able to fly, we have added **enum** Flight as capability to easily distinguish between flying and walking. There are certain behaviours that are exclusive to Pterodactyls in comparison with other Dinosaurs, such as looking for a tree for resting when out of fuel, looking for a tree to lay an egg etc. So we have added PterodactylSeekTree behaviour (**extends** Behavior) which will be used for both resting and laying an egg, following the DrY principle. This behaviour is used in MateBehaviour if ready to lay an egg, and used in the Pterodactyl's playturn method if fuel is over. Since Pterodactyls can be eaten whole by Allosaurs, we have reused the AllosaurAttackBehaviour from assignment 2 to avoid repeating code, again following the DrY principle. Additionally, we have made it possible for Pterodactyls to fly over vending machines, but prevented them from flying over walls.

Sophisticated Game Driver

In order to allow the player to quit, we created a QuitGameAction class that **extends** Action (DRY). This would simply remove the player from the map and display the appropriate message. Similarly, we added a new **Enum** class GameMode, which contains values Challenge and Sandbox, and added this enum as capability to the player, given his choices in the main menu. In challenge mode, once the set number of turns are up, QuitGameAction is called again, allowing us to reuse code.

Priorities

When a dinosaur has multiple needs in the same turn, we have set a priority system as explained below:
For Stegosaurs/Brachiosaurs

1. Mating
2. Laying egg
3. Thirst
4. Hunger

For Allosaurs

1. Mating
2. Laying Egg
3. Attacking (regenerates health points)
4. Thirst
5. Hunger

For Pterodactyls

1. Mating
2. Laying egg
3. Moving towards tree for resting (regaining ability to fly)
4. Moving towards tree for laying egg
5. Thirst
6. Hunger

We have decided that this is the most logical approach for priorities, and also the best method for their survival. For Allosaurs, after mating/laying egg the next best priority is attack(Stegosaur/Pterodactyl), we have done that to make the game more interesting and to increase the importance of the danger of Pterodactyls walking instead of flying, since they would be eaten completely by Allosaurs. For Pterodactyls, due to the previously mentioned reason, we have made resting the next priority after mating/laying eggs.

Things we updated from previous design

Dinosaur Classes

In addition to Stegosaurs, Brachiosaurs and Allosaurs, a new Pterodactyl class was added. In assignment 2, we added a VegetarianDinosaur and CarnivorousDinosaur class as **subclasses** from the **abstract** base Dinosaur class, so that we could add common actions such as FeedVegetarianAction to VegetarianDinosaur, and FeedCarnivoreAction to CarnivoreDinosaur following the Classes should be responsible for their own properties. The new Pterodactyl was added as a child class of CarnivorousDinosaur and therefore it will be able to perform all the actions/behaviours that carnivorous dinosaurs can perform. This follows the DrY principle , since we only needed to add the actions/behaviours/attributes that were exclusive to Pterodactyls, and didn't need to repeat code for shared properties.

Similarly, in assignment 2, we had a class for baby dinosaurs and another class for adult dinosaurs (eg. BabyAllosaur and Allosaur). Since this repeated a lot of the code, we decided to consolidate both classes into one class (DRY principle), and handled age by using the AgeGroup **enum** class. Thus, when a dinosaur of *AgeGroup.BABY* has reached a certain number of turns, we set it to *AgeGroup.ADULT*, and change its display char from a small letter to a capital letter. Eg. (baby Allosaur) *a* → *A* (adult Allosaur). We decided to have separate display characters for baby and adult dinosaurs so it would be more clear to distinguish in the console (for ease of the user)

Dead Dinosaurs and Corpse

We have updated the Corpse class and added a corspePoints attribute to the class instead of creating four new classes for StegosaurCorpse, AllosaurCorpse, PterodactylCorpse and BrachiosuarCorpse because these classes were not necessary since the only common attribute was the amount of points that a carnivore will get by eating this corpse. This is following the ReD principle.

This approach allowed us to ensure that Pterodactyls will only eat a portion of the corpse, so we were able to reduce the corpsePoints by a value 10, each time a Pterodactyl fed from it.

Vending Machine and functionality

We updated the vending machine class such that it will only store the prices of the items that are available for purchase instead of storing an instance of each item, therefore multiple unnecessary dependencies were eliminated, following the ReD principle. All purchases can be done through the BuyItemAction (following classes to be responsible for their own properties principle) which allows the player to purchase an item (shown by a menu) given the player has enough ecoPoints. If so, that item will be **instantiated** and added to the player's inventory. Additionally we have added a brand new PterodactylEgg item for purchase in the VendingMachine.

Players interaction with Dinosaurs

For players to be able to feed dinosaurs, we have created two classes, FeedVegetarianAction and FeedCarnivoreAction since there are two types of dinosaurs in the game. Since one type of dinosaur eats meat,eggs and carnivore meal kits while the other type eats fruits and vegetarian meal kits, having two separate classes would make the code much neater, more structured and allows us to avoid excessive use of literals. Using two classes for feeding instead of the initial plan of having a separate feed action class for each dinosaur allowed us to reduce dependencies (ReD), and the previous design would not be efficient in design terms, since each time a new dinosaur is added to the game, a new feed action class will have to be created, whereas as having two flexible classes is better for future updates to the game.
Regarding controlling the population of Stegosaurs, we have re-used the existing AttackAction for attacking Stegosaurs, to avoid repeating code (DrY) and to minimize dependencies (ReD)

Dinosaur seeking food

Previously we had two separate classes for seeking food, one was SeekMeat and the other was SeekFruit, to reduce dependencies(ReD) we decided to combine these two classes and create a SeekFoodBehaviour class which utilizes the SearchMap described above to search the map for the right type of food for each dinosaur. This combination also allowed us to reduce some repeating code that was used in both classes (DrY). Furthermore, this class will be more flexible for more additions in edible items to the game, aside from just fruits and meat. The SeekFoodBehaviour implements the **Behaviour** interface.

Allosaurs Attacking

We have created an AllosaurAttackBehaviour class that looks through all its adjacent squares by looping through all the exits (integration with the engine) and returns an EatPterodactylAction if a vulnerable Pterodactyl who cannot fly is found which carries out the process of the Alllosaur eating the said Pterodactyl as a whole. If there is an attackable stegosaur (cooldown is over), then an AllosaurAttackAction is returned. Reusing the same AllosaurAttackBehaviour is an example of following the DrY principle

Table Mapping displayChar and objects

Table mapping display characters to important actors, items and GroundType:

Display char	Object
a	Allosaur (Baby)
A	Allosaur (Adult)
r	Brachiosaur (Baby)
R	Brachiosaur (Adult)
s	Stegosaur (Baby)
S	Stegosaur (Adult)
p	Pterodactyl (Baby)
P	Pterodactyl (Adult)
q	StegosaurEgg
w	BrachiosaurEgg
e	AllosaurEgg
y	PterodactylEgg
v	VegetarianMealKit
c	CarnivoreMealKit
C	Corpse
Z	LaserGun
+, t, T	Tree
b	Bush
f	Fruit

~	Lake
#	Wall
_	Floor
X	Vending Machine
@	Player

Git branches

The branches used for each main design feature is listed below(branch name on the right):

- Lakes, Water and Rain - water
- Thirsty Dinosaurs - Assignment3_Amindu(Couldn’t rename)
- Pterodactyls - Pterodactyls
- Second Map - secondMap
- A More Sophisticated Game Driver - newGameDriver